HW#2: Branch Predictor Design

朱致伶 Chu Chih Ling, 110550062

***Abstract*—This report analyze the current branch predictor design method of Aquila SoC, which is an one-level predictor based on bimodal assumption with no collision handling. With analyzing the effectiveness of SoC in several aspects, I proposed a two-level branch predictor method to improve the performance of branch prediction.**

I. Introduction

This homework analyze current one-level branch predictor in following aspect: w/o branch prediction, different size of entry number of Branch History Table (BHT) and hit and miss rate of different branch types of CoreMark. Then, propose the two-level branch predictor method to improve the effectiveness. Lastly, analysis the performance of two-level branch predictor with different sizes of global history and global history pre-filled with 0's.

II. Statistics and Analysis

Most of the signals are caught in execution.v and bpu.v in the bit-stream, running on the FPGA board. Setting diteration parameter to 0, which CoreMark automatically stopped the main function. Analyze current one-level bimodal branch predictor in following aspect:

A. *With/Without branch prediction*

`define ENABLE_BRANCH_PREDICTION in aquila_config.vh controls the enabling and disabling the branch prediction function of CoreMark. If branch prediction is correct, it reduces two clock cycles. However, if branch misprediction happens, it has the same performance as disabling the prediction function. Therefore, Simple bimodal branch prediction does improve performance.

From Table 1, we see that the hit rate of BHT is 84.98%, and the miss rate of branch prediction is 11.74%. In conclusion, (1) Disable the branch prediction function causes more clock cycles to run the Main function, which is low efficient. (2) Main function without branch prediction has less stalls compared to the one with prediction. The potential reasons behind the statistics I guess is that the disability of branch prediction may cause the mistake of stall count.

TABLE I. Comparison With and Without Branch Prediction

Main function with entry_num = 32	*With Branch Prediction*	*Without Branch Prediction*
total_clk_cycle	613907488	681225504
total_stall	166340762	166212934
branch_count	92080344	91952587
hit_count	78253741	x
hit_rate	84.98%	x
misprediction_count	10813379	10813381

Main function with entry_num = 32	*With Branch Prediction*	*Without Branch Prediction*
total_clk_cycle	613907488	681225504
miss_rate	11.74%	x
CoreMark Score	82.433295	74.259713

B. *BHT with different entry sizes*

With the following two equations, we calculate the hit and miss rate of the main function in different entry sizes:

Hit_Rate = Hit Count / Branch Count - *eq1.*

Miss_Rate = Mispred Count / Branch Count - *eq2*

The entry sizes of BHT is crucial to overall performance, since there will be many collisions between different instructions with the same tag.

From Table 2, we get the following conclusion: (1) As the entry sizes increases exponentially from 16 bits to 256 bits, the hit rate of branch history table increases from 77.98% to 97.58%. (2) As for the miss rate, it slightly increases which is an unwanted result. However, compare the increase scale of hit and miss rate, it seems to be unavoidable for the slight increase, since the huge increase of hit rate. (3) As for the CoreMark score, it rises from 81.7382 to 83.2913.

As a result, we assure that the program has better performance with bigger entry sizes of BHT.

TABLE II. Comparison Of BHT With Different Entry Sizes

Main Function	**BHT Entry**				
	16bits	***32 bits***	***64 bits***	***128 bits***	***256 bits***
total_clk_cycle	619108646	613907488	611148702	608876584	607606654
branch_count	92080423	92080344	92080364	92080386	92080403
hit_count	71800517	78253741	84025270	87683444	89852501
hit_rate	77.98%	84.98%	91.25%	95.22%	97.58%
mispred_count	10003582	10813379	11629851	12230080	12455343
miss_rate	10.86%	11.74%	12.63%	13.28%	13.53%
score	81.73820	82.433295	82.806783	83.116946	83.291303

C. *Hit and Miss Rate of Different Branch Types*

Table 3 records the hit, miss rate and distribution of different branch types in entry_num=32 bits.

TABLE III. COMPARISON OF DIFFERENT BRANCH TYPES

Main Function entry=32	Branch Type				
	branch _count	*hit _count*	*hit_ rate*	*mispred _count*	*mispred _rate*
beq	36056408	28982516	80.38%	5796607	16.07%
bne	35499483	33543449	94.48%	4195035	11.81%
blt	1040861	881051	84.64%	149835	14.40%
bge	2280575	2120745	92.99%	296738	13.01%
bltu	3015697	2601748	86.27%	213010	7.06%
bgeu	14187347	2120746	14.95%	162159	1.14%
total	92080444	78253836	84.98%	10813384	11.74%

From Table 3, we observe the following symptoms: (1) The two following branch types: beq, bne make up a large portion of branches count. (2) Most branch types have high hit rate of BHT above 80% except bgeu. (3) The average misprediction rate of all branch types is around 12%.

III. TWO-LEVEL BRANCH PREDICTOR DESIGN CONCEPT

Based on previous analysis, we get the 3 following conclusions: (1) CoreMark has better performance with branch prediction. (2) As the entry sizes of the BHT increases, the performance also increases. (3) beq and bne are common branch types in CoreMark.

After analyze the current branch prediction method, I would like to implement the two-level branch prediction to improve the performance. The following are the brief conclusion of mine with the three types of predictor used in the design of improvement:

A. Local Branch Predictor

The Local Branch Predictor requires two tables: Pattern History Table (PHT) and Saturating Counter (bimodal).

Pattern History Table is specific to a particular branch. It records the historical behavior of the corresponding **`pc_addr`** (n bits). As for Saturating Counter (bimodal), which is also the branch likelihood. When a branch hit occurs, it first uses the **`pc_addr`** (n bits) to search the PHT for the corresponding pattern (m bits). This pattern is then used as an index to access the saturating counter to obtain the corresponding prediction output.

When `m` is sufficiently large and doesn't coincide with branches having the same pattern, this approach accurately predicts the type of branch with a fixed pattern. In contrast, a typical bimodal predictor would make incorrect predictions for one-third of the branches. The disadvantage is that when two branches with the same pattern but opposite decisions occur, they might interfere with each other as they share the same counter entry.

B. Global Branch Predictor

The Global Branch Predictor considers only the global history pattern. When a branch instruction is executed, it uses the taken and not taken history of various branch types as an index to fetch the prediction result from a table. The downside is that if the table size is too small, different branch instructions are likely to share the same table entry.

C. Two-Level Branch Predictor

I implement gshare two-level branch predictor, since using global accuracy alone tends to be lower, the effectiveness can be worse than bimodal in the case of a small table size.

The approach of gshare allows the branch predictor to consider both global history and the current **`pc_addr`** as the index bits simultaneously. Gshare approach performs an XOR operation on **`pc_addr`** and branch history (n bits) to obtain an index bit for the bimodal table to get the prediction result.

IV. IMPLEMENT AND ANALYZE TWO-LEVEL BRANCH PREDICTION

To implement the previous two-level branch prediction method, I implement the gshare predictor in bpu.v.

A. GSHARE Method

The main concept of gshare is implementing an XOR operation with global history and instruction address before looking up the pattern table. Therefore, I modified the write and read address of the branch_likelihood table as the following code:

```
read_addr = (pc_i[NBITS+2:2] ^ global_history[NBITS-1:0])
write_addr = (dec_pc_i[NBITS+2:2] ^ global_history[NBITS-1:0])
```

B. GSHARE – global history in different bits

The bits of global history originally depends on the entry sizes of BHT, since the index of BHT, which is the "tag", is in NBITS = $clog2(ENTRY_NUM) bits. Therefore, I came up with the idea of testing the gshare performance with global history in different bits.

TABLE IV. COMPARISON OF GLOBAL HISTORY IN DIFFERENT SIZES

Global History Size	CoreMark Score	hit rate	misprediction rate
Prediction off	74.259713	X	X
4 bits	79.629800	46.42%	6.56%
5 bits	79.804100	50.96%	7.25%
6 bits	79.960421	56.26%	8.18%
7 bits	80.169493	59.37%	8.75%
8 bits	80.233406	61.13%	8.98%
9 bits	80.834960	62.71%	7.35%
Saturating Counter (9 bits)	83.759323	99.21%	13.78%

*Compare the marco-results of gshare and saturating counter, **gshare has lower hit rate and higher miss rate**. I speculate the reason behind the synonym is the increases of possible tag cases of BHT, caused by the XOR operation of global history and instruction address. Since the tag index seldom duplicate due to the various possible cases, the decision of branch prediction has higher accuracy whenever it hits.*

From Table 4, we can see (1) The CoreMark score increases as the global history sizes increases, which follows the conclusion from the performance comparison with different global history sizes. (2) As the global history sizes grow, the increase scale of hit rate is larger than the scale of miss rate, lead to growing CoreMark score. (3) The hit rate increases sharply in 16, 32, 64 bits, and gradually rises in 128, 256, 512 bits, which gets steady at around 60%. (3) The miss rate slightly increases, however, unexpectedly drop to 7.35% at 512 bits.

As you can noticed from Table 4, the gshare method doesn't enhance the performance as we expected, instead causing a drop on CoreMark score. I speculate the reasons behind the unwanted low performance is the unnecessary of long bit global history. For example, there are three function A, B and C. Function A and B may jump to function C. However, function A and B need three and five branches respectively to jump to function C. With the current gshare mechanism, it always uses 5 bits of global history to calculate XOR address and take it as the index of BHT. The above scenario may lead to misprediction.

V. Other Important Things to Discuss

A. BHT Hit or Branch Prediction Hit

In this homework, I was confused whether to count the hit rate of "BHT table hit" or "BPU prediction(decision) hit". Indeed, these two are worth analyzing, but I think it is more essential to look into the BHT table hits.

The BHT table hit rate records the frequency of whether the current branch prediction is based on previous prediction, which is an essential reference for branch prediction. If the current branch doesn't hit the BHT, it may predict with no reference, causing to bad performance.

As for the BPU prediction(decision) hit, it is a bit similar to the misprediction rate, which is counted in the homework. Misprediction rate stores the rate of final prediction's correctness, which is calculated with two previous count: the BPU decision and taken/non-taken final result.

B. Combination of Gshare and Saturating Counter

Since the CoreMark score of Gshare method is lower than saturating counter, I tried to combine the two branch prediction method, wondering whether the combination gets better performance. As the consequence, the CoreMark score of the combination of two method is 84.37922 with entry size = 512, global history bits = 9 with 4 bits of pre-filled 0's, which out-performs as expected.

Although, it has the highest score among all previous method, it doesn't imply it has the best overall-performance, since the area of BPU on FPGA board becomes large. The large area of unit on board may have high energy consumption and heat dissipation problem, which are aspects we didn't emphasize in this assignment. The combination of gshare and saturating counter making BPU the largest unit on board, the high CoreMark score company with other issues as the side effect. Therefore, I didn't take a deeper look of the performance of combination method.

C. Saturating Counter over Two-Level Branch Predictor

After conducting some research, it was observed that some argue that saturating counters are generally more power-efficient due to their involvement in fewer complex computations. Additionally, they are considered space-efficient since they can be implemented with a small number of flip-flops.

However, there are compelling reasons to explore alternative approaches for enhancing predictive performance. One noteworthy limitation is that this one-level predictor does not take into account the results of other branch instructions or the global history. Moreover, collisions can arise when two distinct branch instructions share the same least significant bits, and resolving this issue necessitates an increase in entries in the Branch History Table (BHT). While direct mapping solely requires wiring, modern computers employ associative caches for various reasons.

VI. Summary

It takes a lot of efforts for me to complete this assignment. Since the detailed structure of Branch predicting is definitely a new knowledge for me to pick up, which is not taught in the computer organization course, so understanding the whole data-flows takes me a lot of time.

In this homework, I analyze the CoreMark performance in following aspects: with and without branch prediction, different entry sizes of BHT, different branch types and gshare in different size of global history bits. Although I did not configure out a powerful way to enhance the overall branch predicting performance, but at least I get to know them better. With this practice, I get to know why branch predicting has been a widely-discussed topic over a long time.